语义分析-实验报告

2021201709 李俊霖

在词法和语法实验的基础上，添加、修改语义动作，完成语义分析，并输出`SysY`代码对应的`x86 AT&T`汇编代码。

词法和语法分析部分已在前两个实验中展示，故不在此赘述。

0. 项目文件介绍

- `word.l`为词法分析文件
- `grammar.y`为语法、语义分析文件
- `run.sh`为运行脚本
- `result`文件夹中
  - `test`和`1-5`文件夹分别存放有`.sy`文件和生成的汇编代码。

1. 数据结构

- 全局变量，用于存储某个时刻的节点个数、汇编语句条数、全局偏移量、 嵌套层数、函数名等，便于在语法分析、语义分析过程中进行移进和归约。

```
int cnt = 0; //节点个数
int num_of_assem = 0; //汇编语句条数
int offset = 0; //全局偏移量
int level = 0; //层数（嵌套）
char L[100000][50]; //lable
char f_name[100]; //函数名
```

- 属性节点：记录每个归约的符号的状态，所有的节点共用该相同的数据结构，便于管理。对于任意一个节点，并非该数据结构中的所有变量都会被使用：如常数节点不存在`truelist、falselist`。

```
// 属性节点
struct State{
    int value, offset, offset_of_arr;
    int quad;
    bool is_const;
    bool is_arr;
    string name;
    vector<bool> const_para, arr_para; //参数类型
    vector<int> truelist, falselist;
    vector<int> dimension;
    vector<int> para_val, para_offset, para_offset_of_arr;
    vector<string> para_name;
}node[100000];
```

- 符号表：使用map直接存储，便于直接从符号的名称查找到该符号的结构体信息。

```
vector<map<string, Variable> > symbol_table; //符号表
```

- 变量结构体（Variable）：存储符号的信息，如类型、值、偏移量、每一层的维数（对于数组）。

```
//变量结构体，存储变量的信息
struct Variable{
    Type type; //类型
    int value; //值，当且仅当为常量时，该值有效
    int offset; //偏移
    vector<int> dimension; //维数，当且仅当该变量为数组时，该值有效
    Variable() {}
    //构造函数
    Variable(Type _type, int _val) : type(_type), value(_val){}
    Variable(Type _type, int _val, int _offset) : type(_type), value(_val),
offset(_offset){}
    Variable(Type _type, int _val, int _offset, vector<int> _dim) : type(_type),
value(_val), offset(_offset), dimension(_dim){}
};
```

- 其他特殊数据和结构（用途见注释）

```
vector<string> Assemble_code; //存储汇编指令
vector<string> Parameter; //存储参数，参数列表
vector<vector<pair <int, int> > > breaklist, continuelist; //存储break和continue语
句的地址
```

2. 寄存器和栈的说明和基本操作

2.1. 寄存器

- `%rsp`、`%rbp` 保留作为栈的栈顶、栈底指针。
- 表达式运算时（加减乘法）用`%r8`，`%r9`存储左右操作数进行计算，存入`%r8`中。
- 除法和求余运算，被除数存入`%eax`中。
- 在索引数组中的值时偏移量固定用`%rbx` `%rdx`来计算（不用`%eax`，否则当需要用该值作为除法/求余运算的分母时，会产生冲突）。
- 函数的返回值存入`%eax`中。
- 此外函数传参存储在栈上，不保存在寄存器中。

2.2. 栈

- 局部变量（变量和数组）保存在栈上。
- 函数传递的参数保存在栈上。
- 函数进入时，`pushq`保存寄存器，即将`callee-saved`的寄存器push到栈上返回时，`popq`弹出寄存器，恢复栈。

2.3. 变量->寄存器，寄存器->变量

- 使用如下函数实现变量和寄存器之间的转换存储。

```
void var_reg(int x, const char* reg);
void reg_var(const char* reg, int x);
```

- 以将变量存入寄存器为例：
    - 如果为常量，则直接 mov 数值。
    - 如果为全局变量或数组，则直接用 rip 寄存器和变量名定位。
    - 如果为局部变量，则用栈内偏移量定位。
    - 以下代码以全局数组为例：

```
//将变量移入寄存器
if(node[x].is_arr){//数组，用 rip 寄存器和变量名定位
    sprintf(tmp, "\tmovl\t%d(%rbp), %%ebx\n", node[x].offset_of_arr);
    Assemble.push_back(tmp);
    Assemble.push_back("\tcltq\n");
    Assemble.push_back("\tleaq\t0(, %rbx, 4), %rdx\n");
    sprintf(tmp, "\tleaq\t%s(%rip), %rbx\n", node[x].name.c_str());
    Assemble.push_back(tmp);
    sprintf(tmp, "\tmovl\t(%rdx, %rbx), %%%s\n", reg);
    Assemble.push_back(tmp);
}
```

3. 常量处理

- 需要满足赋值必须是常数。
- 对于定义在全局的常量，将常数定义在汇编代码的全局区域，标记为常数（`const_para`），直接获得该变量的值，在后续的计算中直接用这个值来计算。
- 局部定义的常量定义在栈上。

以下代码以定义在全局区域的常量为例展示：

```
//常量的定义，满足赋的值必须是常数
if (!$2){
    Assemble.push_back("\t.section\t.rodata\n");
    Assemble.push_back("\t.align\t4\n");
    sprintf (tmp, "\t.type\t%s, @object\n", L[$1]);
    Assemble.push_back(tmp);
    sprintf (tmp, "\t.size\t%s, 4\n", L[$1]);
    Assemble.push_back(tmp);
    sprintf (tmp, "%s:\n", L[$1]);
    Assemble.push_back(tmp);
    sprintf (tmp, "\t.long\t%d\n", node[$4].value);
    Assemble.push_back(tmp);
    Assemble.push_back("\t.text\n");
    symbol_table[0][L[$1]] = Variable(Int_Const, node[$4].value, 1);
}
```

4. 变量处理

- 全局变量统一放`.data`段，局部变量统一放在栈上。
- 若对该变量只定义不赋值：
  - 对于全局变量，计算相应的大小 size，对于变量初值设为.long 0 或.zero 4。
  - 对于全局数组，对于数组初始化值为.zero 4 * size。
  - 对局部变量，在栈上分配空间，初始化为零即可。
- 若对该变量赋初值：
  - 赋的初值要为常数。
  - 对于全局变量，根据`InitVal`的返回值情况判断类型是否匹配。
  - 对于全局数组，判断赋值的个数是否超过数组的大小。
  - 对于局部变量，赋的初值可能不是常数，需要通过函数获取变量的值，然后存在栈上相应的位置。
  - 数组使用右递归进行归约。

以有赋初值的数组为例：

```
//全局数组，判断赋值的个数是否超过了数组大小
if($4){
    yyerror("Initializer Error");
    exit(0);
}
if(ArrInitVal.size() > node[$2].value){
    yyerror("Too Many Initializers Error");
    exit(0);
}
sprintf (tmp, "\t.globl\t%s\n", L[$1]); Assemble.push_back(tmp);
Assemble.push_back("\t.data\n");
Assemble.push_back("\t.align\t32\n");
sprintf (tmp, "\t.type\t%s, @object\n", L[$1]); Assemble.push_back(tmp);
sprintf (tmp, "\t.size\t%s, %d\n", L[$1], node[$2].value * 4);
Assemble.push_back(tmp);
sprintf (tmp, "%s:\n", L[$1]); Assemble.push_back(tmp);
for(auto x : ArrInitVal) {
    sprintf (tmp, "\t.long\t%d\n", node[x].value);  Assemble.push_back(tmp);
}
sprintf (tmp, "\t.zero\t%d\n", node[$2].value * 4 - ArrInitVal.size() * 4);
Assemble.push_back(tmp);
Assemble.push_back("\t.text\n");
symbol_table[0][L[$1]] = Variable(Array, 0, 1, node[$2].dimension);
ArrInitVal.clear();
```

5. 函数调用

- 对于参数部分，在匹配到函数名后， 匹配参数前将参数列表清空
- 每次匹配到一个参数时将其 push_back 到参数列表中
- 进入函数（以int函数为例）：查看每一层是否有重名函数，创建符号表，进入新的嵌套层，新开一个函数体，进入函数pushq三件套。

```
Entry_Int: /*empty*/ {
```

```
    //查看每一层是否有重名函数
    for(int i = level; i >= 0; --i)
        if(symbol_table[i].find(funcname) != symbol_table[i].end()){
            yyerror("Function Redefinition Error");
            exit(0);
        }
    //创建符号表
    symbol_table[level][funcname] = Variable(int_Function, 0, 0);
    //进入新的嵌套层
    ++level;
    map<string, Variable> x;
    symbol_table.push_back(x);
    char tmp[100];
    //新开一个函数体
    sprintf(tmp, "\t.globl\t%s\n", funcname); Assemble.push_back(tmp);
    Assemble.push_back("\t.type\tmain, @function\n");
    sprintf(tmp, "%s:\n", funcname); Assemble.push_back(tmp);
    //进入函数pushq三件套
    call_func_push();
}
;
```

- 函数结束归约时：退出该层嵌套，符号表弹出，将栈恢复，弹出旧寄存器值

```
//函数结束归约时，将栈恢复，弹出旧寄存器值
FuncDef:
| INT FName '(' FuncFParams ')' Entry_Int_Para Block{
    //退出该层嵌套
    --level;
    //符号表弹出
    symbol_table.pop_back();
    char tmp[100];
    //将栈恢复
    sprintf(tmp, "\taddq\t$%d, %rsp\n", -offset);
    Assemble.push_back(tmp);
    //弹出旧寄存器值
    ret_func_pop();
    offset = 0;
}
```

- 参数的匹配使用右递归，每匹配到一个参数时将参数存到当前的参数列表中。

```
FuncFParams: FuncFParam {}
    | FuncFParams ',' FuncFParam{}
    ;

FuncFParam: INT ID ParaArr{Para.push_back(L[$2]);}
    ;
```

6. 数组

- 匹配时记录数组每一维的维数，统计数组所需要占据的空间大小，使用dimension的容器记录每一维的大小。
- 使用时，递归从最后一维开始计算该变量所在的地址，一步一步使用汇编实现地址的计算。
- 数组作为函数参数传递即数组地址，可以直接在符号表中进行记录即可。

```
Array:  /*empty*/ {$$ = 0;}
    | '[' Exp ']' Array{
        $$ = ($4) ? $4 : ++cnt;
        node[$$].dimension.push_back($2);
    }
    ;
```

7. 控制流语句

- 为了便于进行归约、回填和合并，将控制流语句的语法修改如下：

```
//语法修改：
IF '(' Cond ')' NewLabel BeforeStmt Stmt AfterStmt %prec WITHOUTELSE
IF '(' Cond ')' NewLabel BeforeStmt Stmt AfterStmt ELSE AfterElse NewLabel
BeforeStmt Stmt AfterStmt NewLabel
WHILE WhileBegin BeforeStmt '(' Cond ')' WhileEnd NewLabel Stmt AfterStmt
```

- `if`语句：
  - 在`NewLable`处，将`NewLable`的首地址回填到条件判断的`truelist`和`falselist`中。
  - ```
    Stmt:
        //if语句2
        | IF '(' Cond ')' NewLabel BeforeStmt Stmt AfterStmt ELSE AfterElse
    NewLabel BeforeStmt Stmt AfterStmt NewLabel{
            --level;
            symbol_table.pop_back();
            for(auto x : node[$3].truelist) Assemble[x] += ".L" +
    to_string($5) + "\n";
            for(auto x : node[$3].falselist) Assemble[x] += ".L" +
    to_string($11) + "\n";
            Assemble[node[$10].truelist[0]] += ".L" + to_string($15) + "\n";
        }
    ```

- `while`语句。
  - 进入while语句之前，标记每趟循环判断语句之前的位置， 新建一个标签和break/continue list 层。
    ```

- ```
  //标记每趟循环判断语句之前的位置， 新建一个标签和break/continue list 层
  WhileBegin: /*empty*/ {
          $$ = ++num_of_assem;
          char tmp[100];
          sprintf (tmp, ".L%d:\n", num_of_assem); Assemble.push_back(tmp);
          vector< pair<int, int> > x, y;
          breaklist.push_back(x);
          continuelist.push_back(y);
      }
      ;
  ```

- 在while语句归约时，在有`breaklist`的地方加上无条件跳转，恢复栈内容后跳转

- ```
  //在有breaklist的地方加上无条件跳转，恢复栈内容后跳转
          for(auto it : *breaklist.rbegin()){
              sprintf(tmp, "\taddq\t$%d, %rsp\n", offset - it.second);
              Assemble[it.first - 1] = string(tmp);
              Assemble[it.first] += ".L" + to_string(num_of_assem) + "\n";
          }
  ```

- 完整版代码见`grammar.y`文件。

8. 运行结果展示

按顺序分别为`test.sy, 1.sy, 2.sy, 3.sy, 4.sy`，输出结果正确。

```
ljl@LAPTOP-SFK1PRFG:/mnt/g/2023 spring semester/Compilers/lab/semantic/3-语义实验/src-final$ ./run.sh
233
ljl@LAPTOP-SFK1PRFG:/mnt/g/2023 spring semester/Compilers/lab/semantic/3-语义实验/src-final$ ./run.sh
18 10
28
ljl@LAPTOP-SFK1PRFG:/mnt/g/2023 spring semester/Compilers/lab/semantic/3-语义实验/src-final$ ./run.sh
2333
3628810
0
2
5
3
8
5
2
520
520
ljl@LAPTOP-SFK1PRFG:/mnt/g/2023 spring semester/Compilers/lab/semantic/3-语义实验/src-final$ ./run.sh
2 2
1 1
1 1
1
2
1
ljl@LAPTOP-SFK1PRFG:/mnt/g/2023 spring semester/Compilers/lab/semantic/3-语义实验/src-final$ ./run.sh
6
```

9. 总结与思考

1. 由于笔者的时间有限，代码未做编译过程的错误处理。
2. 为了更高效地完成编译器的编写，笔者在进行独立思考后，参考了前人一些实现起来比较好的思路和策略，如寄存器的分配和数据结构的组织。但限于学业压力，本编译器实现的功能有限，可以完成基本的功能要求和实验任务，如对于复杂类型的数据结构（结构体、枚举类型）等无法编译，这是后期完善时可以改进之处。

3. 实现上为了方便，许多功能的实现只考虑了完成度，效率较低。比如函数传参时，笔者将参数存储在栈上，不保存在寄存器中，当需要对参数进行提取、运算等操作时，需要从栈中提取出，效率不高；另外，寄存器分配的策略比较单一，没有采用复杂的算法去获取最优的寄存器，编译出来的程序运行效率也不高。以上都是可以改进的地方。